6 ORDENAÇÃO POR HEAP

Neste capítulo, introduzimos outro algoritmo de ordenação: ordenação por heap. Como a ordenação por intercalação, mas diferente da ordenação por inserção, o tempo de execução da ordenação por heap é $O(n \lg n)$. Como a ordenação por inserção, mas diferentemente da ordenação por intercalação, a ordenação por heap ordena no lugar: apenas um número constante de elementos do arranjo é armazenado fora do arranjo de entrada em qualquer instante. Assim, a ordenação por heap combina os melhores atributos dos dois algoritmos de ordenação que já discutimos.

A ordenação por heap também introduz outra técnica de projeto de algoritmos: a utilização de uma estrutura de dados, nesse caso uma estrutura que denominamos "heap" para gerenciar informações. A estrutura de dados heap não é útil apenas para a ordenação por heap, ela também cria uma eficiente fila de prioridades. A estrutura de dados heap reaparecerá em algoritmos em capítulos posteriores.

O termo "heap" foi cunhado originalmente no contexto da ordenação por heap, mas desde então passou a se referir também a "armazenamento com coleta de lixo", tal como dado pelas linguagens de programação Lisp e Java. Nossa estrutura de dados heap *não* é armazenamento com coleta de lixo e, sempre que mencionarmos heaps neste livro, o termo significa a estrutura de dados definida neste capítulo.

6.1 HEAPS

A estrutura de dados ***heap*** (***binário***) é um objeto arranjo que pode ser visto como uma árvore binária quase completa (veja Seção B.5.3), como mostra a Figura 6.1. Cada nó da árvore corresponde a um elemento do arranjo. A árvore está completamente preenchida em todos os níveis, exceto possivelmente no nível mais baixo, que é preenchido a partir da esquerda até um ponto. Um arranjo A que representa um heap é um objeto com dois atributos: $A \cdot comprimento$, que (como sempre) dá o número de elementos no arranjo, e $A \cdot tamanho\text{-}do\text{-}heap$, que representa quantos elementos no heap estão armazenados dentro do arranjo A. Isto é, embora $A[1 \,..\, A \cdot comprimento]$ possa conter números, só os elementos em $A[A \cdot tamanho\text{-}do\text{-}heap]$, onde $A \cdot tamanho\text{-}do\text{-}heap \leq A \cdot comprimento$, são elementos válidos do heap. A raiz da árvore é $A[1]$ e, dado o índice i de um nó, podemos calcular facilmente os índices de seu pai, do filho à esquerda e do filho à direita:

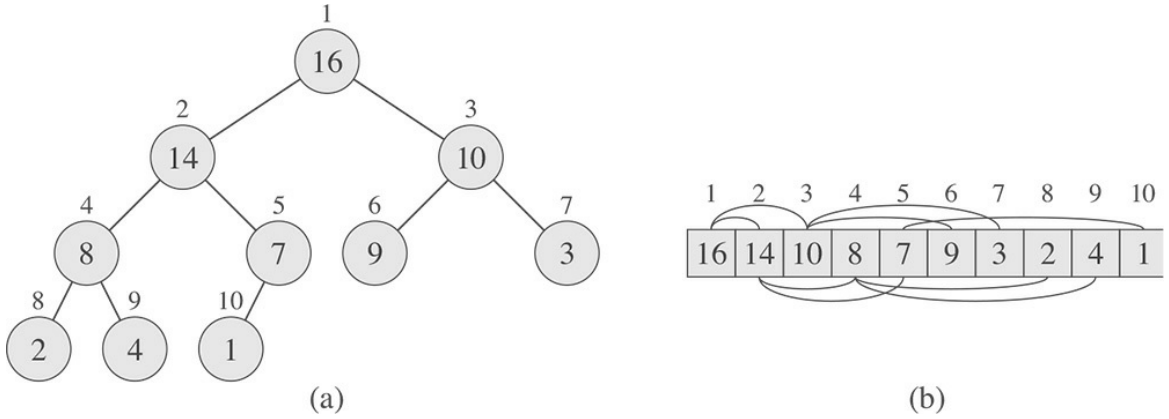

Figura 6.1 Um heap de máximo visto como (**a**) uma árvore binária e (**b**) um arranjo. O número dentro do círculo em cada nó na árvore é o valor armazenado nesse nó. O número acima de um nó é o índice correspondente no arranjo. Acima e abaixo do arranjo há linhas que mostram relacionamentos pai-filho; os pais estão sempre à esquerda de seus filhos. A árvore tem altura três; o nó no índice 4 (com valor 8) tem altura um.

```
PARENT(i)
1  return ⌊i/2⌋

LEFT(i)
1  return 2i

RIGHT(i)
1  return 2i + 1
```

Na maioria dos computadores, o procedimento LEFT pode calcular $2i$ em uma única instrução, simplesmente deslocando a representação binária de i uma posição de bit para a esquerda. De modo semelhante, o procedimento RIGHT pode calcular rapidamente $2i + 1$ deslocando a representação binária de i uma posição de bit para a esquerda e depois inserindo 1 como o bit de ordem baixa. O procedimento PARENT pode calcular $i/2$ deslocando i uma posição de bit para a direita. Uma boa implementação de ordenação por heap frequentemente implementa esses procedimentos como "macros" ou "em linha".

Existem dois tipos de heaps binários: heaps de máximo e heaps de mínimo. Em ambos os tipos, os valores nos nós satisfazem a uma ***propriedade de heap***, cujos detalhes específicos dependem do tipo de heap. Em um ***heap de máximo***, a ***propriedade de heap de máximo*** é que, para todo nó i exceto a raiz,

$$A[\text{PARENT}(i)] \geq A[i] ,$$

isto é, o valor de um nó é, no máximo, o valor de seu pai. Assim, o maior elemento em um heap de máximo é armazenado na raiz, e a subárvore que tem raiz em um nó contém valores menores que o próprio nó. Um ***heap* de *mínimo*** é organizado de modo oposto; a ***propriedade de heap de mínimo*** é que, para todo nó i exceto a raiz,

$$A[\text{PARENT}(i)] \leq A[i] .$$

O menor elemento em um heap de mínimo está na raiz.

Para o algoritmo de ordenação por heap, usamos heaps de máximo. Heaps de mínimo são comumente empregados em filas de prioridades, que discutiremos na Seção 6.5. Seremos precisos ao especificar se necessitamos de um heap de máximo ou de um heap de mínimo para qualquer aplicação particular e, quando as propriedades se aplicarem tanto a heaps de máximo quanto a heaps de mínimo, simplesmente usaremos o termo "heap".

Visualizando um heap como uma árvore, definimos a ***altura*** de um nó em um heap como o número de arestas no caminho descendente simples mais longo desde o nó até uma folha e definiremos a altura do heap como a altura de sua

raiz. Visto que um heap de n elementos é baseado em uma árvore binária completa, sua altura é $\Theta(\lg n)$ (veja Exercício 6.1-2). Veremos que as operações básicas em heaps são executadas em tempo que é, no máximo, proporcional à altura da árvore e, assim, demoram um tempo $O(\lg n)$. O restante deste capítulo apresenta alguns procedimentos básicos e mostra como eles são usados em um algoritmo de ordenação e uma estrutura de dados de fila de prioridades.

- O procedimento MAX-HEAPIFY, que roda no tempo $O(\lg n)$, é a chave para manter a propriedade de heap de máximo (6.1).
- O procedimento BUILD-MAX-HEAP, executado em tempo linear, produz um heap de máximo a partir de um arranjo de entrada não ordenado.
- O procedimento ordenação por heap, executado no tempo $O(n \lg n)$, ordena um arranjo no lugar.
- Os procedimentos MAX-HEAP-INSERT, HEAP-EXTRACT-MAX, HEAP-INCREASE-KEY e HEAP--MAXIMUM, que rodam em tempo $O(\lg n)$, permitem que a estrutura de dados heap implemente uma fila de prioridades.

Exercícios

6.1-1 Quais são os números mínimo e máximo de elementos em um heap de altura h?

6.1-2 Mostre que um heap de n elementos tem altura $\lg n$.

6.1-3 Mostre que, em qualquer subárvore de um heap de máximo, a raiz da subárvore contém o maior valor que ocorre em qualquer lugar nessa subárvore.

6.1-4 Em que lugar de heap de máximo o menor elemento poderia residir, considerando que todos os elementos sejam distintos?

6.1-5 Um arranjo que está em sequência ordenada é um heap de mínimo?

6.1-6 A sequência (23, 17, 14, 6, 13, 10, 1, 5, 7, 12) é um heap de máximo?

6.1-7 Mostre que, com a representação de arranjo para ordenar um heap de n elementos, as folhas são os nós indexados por $n/2 + 1, n/2 + 2, \ldots, n$.

6.2 MANUTENÇÃO DA PROPRIEDADE DE HEAP

Para manter a propriedade de heap de máximo, chamamos o procedimento MAX-HEAPIFY. Suas entradas são um arranjo A e um índice i para o arranjo. Quando chamado, MAX-HEAPIFY considera que as árvores binárias com raízes em LEFT(i) e RIGHT(i) são heaps de máximo, mas que $A[i]$ pode ser menor que seus filhos, o que viola a propriedade de heap de máximo. MAX-HEAPIFY permite que o valor em $A[i]$ "flutue para baixo" no heap de máximo, de modo que a subárvore com raiz no índice i obedeça à propriedade do heap de máximo.

```
MAX-HEAPIFY(A, i)
 1  l = LEFT(i)
 2  r = RIGHT(i)
 3  if l ≤ A·tamanho-do-heap e A[l] > A[i]
 4      maior = l
 5  else maior = i
 6  if r ≤ A·tamanho-do-heap e A[r] > A[maior]
 7     maior = r
 8  if maior ≠ i
 9        trocar A[i] com A[maior]
10        MAX-HEAPIFY(A, maior)
```

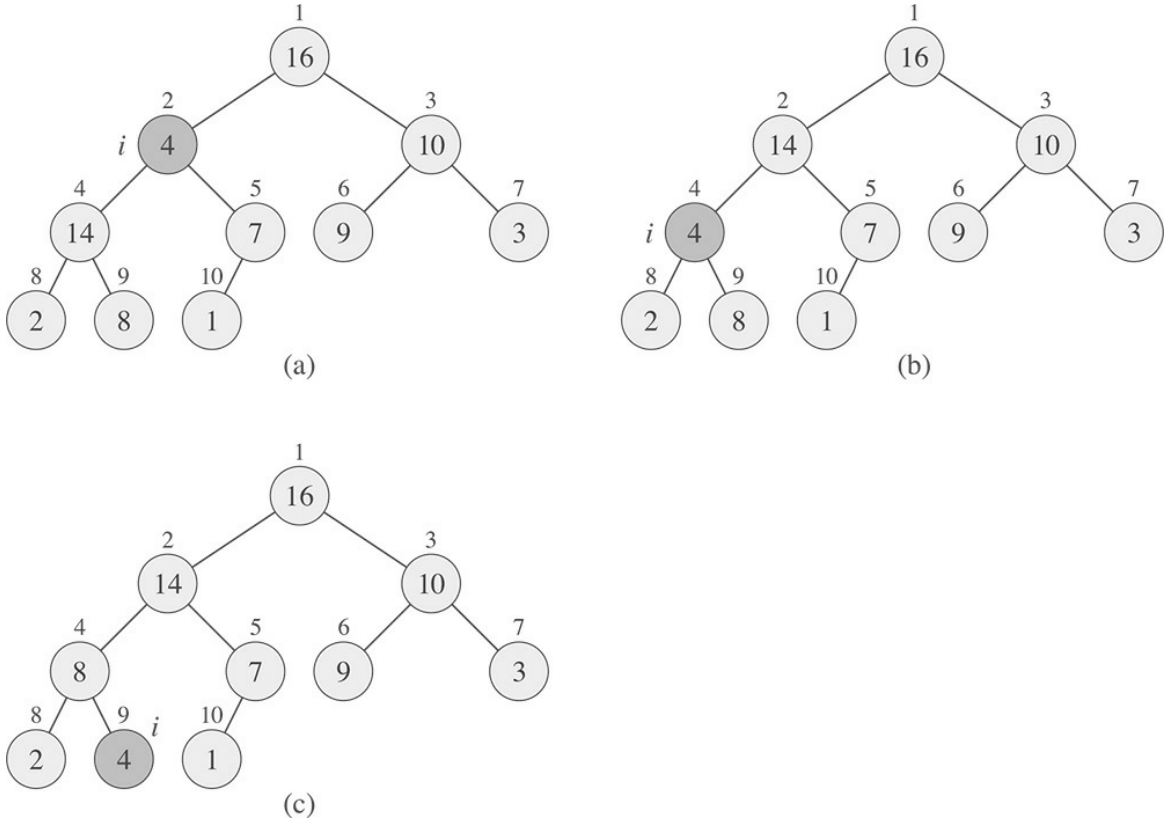

Figura 6.2 Ação de MAX-HEAPIFY(A, 2), onde $A \cdot tamanho\text{-}do\text{-}heap\ [A] = 10$. (**a**) Configuração inicial, com $A[2]$ no nó $i = 2$, violando a propriedade de heap de máximo, já que ele não é maior que os filhos. A propriedade de heap de máximo é restabelecida para o nó 2 em (**b**) pela troca de $A[2]$ por $A[4]$, o que destrói a propriedade de heap de máximo para o nó 4. A chamada recursiva MAX-HEAPIFY(A, 4) agora tem $i = 4$. Após trocar $A[4]$ por $A[9]$, como mostramos em (**c**), o nó 4 é corrigido, e a chamada recursiva a MAX-HEAPIFY(A, 9) não produz nenhuma mudança adicional na estrutura de dados.

A Figura 6.2 ilustra a ação de MAX-HEAPIFY. Em cada etapa, o maior dos elementos $A[i]$, $A[\text{LEFT}(i)]$ e $A[\text{RIGHT}(i)]$ é determinado, e seu índice é armazenado em *maior*. Se $A[i]$ é maior, a subárvore com raiz no nó i já é um heap de máximo, e o procedimento termina. Caso contrário, um dos dois filhos tem o maior elemento, e $A[i]$ é trocado por $A[maior]$, fazendo com que o nó i e seus filhos satisfaçam a propriedade de heap de máximo. Porém, agora o nó

indexado por *maior* tem o valor original $A[i]$ e, assim, a subárvore com raiz em *maior* poderia violar a propriedade de heap de máximo. Em consequência disso, chamamos MAX-HEAPIFY recursivamente nessa subárvore.

O tempo de execução de MAX-HEAPIFY em uma subárvore de tamanho n com raiz em um dado nó i é o tempo Q(1) para corrigir as relações entre os elementos $A[i]$, $A[\text{LEFT}(i)]$ e $A[\text{RIGHT}(i)]$, mais o tempo para executar MAX-HEAPIFY em uma subárvore com raiz em um dos filhos do nó i (considerando que a chamada recursiva ocorre). Cada uma das subárvores dos filhos tem, no máximo, tamanho igual a $2n/3$ — o pior caso ocorre quando a última linha da árvore está exatamente metade cheia — e, portanto, podemos descrever o tempo de execução de MAX-HEAPIFY pela recorrência

$$T(n) \leq T(2n/3) + \Theta(1) .$$

A solução para essa recorrência, de acordo com o caso 2 do teorema mestre (Teorema 4.1), é $T(n) = O(\lg n)$. Como alternativa, podemos caracterizar o tempo de execução de MAX-HEAPIFY em um nó de altura h como $O(h)$.

Exercícios

6.2-1 Usando a Figura 6.2 como modelo, ilustre a operação de MAX-HEAPIFY$(A, 3)$ sobre o arranjo $A = \langle 27, 17, 3, 16, 13, 10, 1, 5, 7, 12, 4, 8, 9, 0\rangle$.

6.2-2 Começando com o procedimento MAX-HEAPIFY, escreva pseudocódigo para o procedimento MIN-HEAPIFY(A, i), que executa a manipulação correspondente sobre um heap de mínimo. Compare o tempo de execução de MIN-HEAPIFY com o de MAX-HEAPIFY.

6.2-3 Qual é o efeito de chamar MAX-HEAPIFY(A, i) quando o elemento $A[i]$ é maior que seus filhos?

6.2-4 Qual é o efeito de chamar MAX-HEAPIFY(A, i) para $i > A \cdot \textit{tamanho-do-heap}/2$?

6.2-5 O código para MAX-HEAPIFY é bastante eficiente em termos de fatores constantes, exceto possivelmente para a chamada recursiva na linha 10, que poderia fazer com que alguns compiladores produzissem código ineficiente. Escreva um MAX-HEAPIFY eficiente que use um constructo de controle iterativo (um laço) em vez de recursão.

6.2-6 Mostre que o tempo de execução do pior caso de MAX-HEAPIFY para um heap de tamanho n é W(lg n). (*Sugestão*: Para um heap com n nós, dê valores de nós que façam MAX-HEAPIFY ser chamado recursivamente em todo nó em um caminho desde a raiz até uma folha.)

6.3 CONSTRUÇÃO DE UM HEAP

Podemos usar o procedimento MAX-HEAPIFY de baixo para cima para converter um arranjo $A[1 .. n]$, onde $n = A \cdot \textit{comprimento}$, em um heap de máximo. Pelo Exercício 6.1-7, os elementos no subarranjo $A[([n/2] + 1) .. n]$ são folhas da árvore e, portanto, já de início, cada um deles é um heap de 1 elemento. O procedimento BUILD-MAX-HEAP percorre os nós restantes da árvore e executa MAX-HEAPIFY sobre cada um.

```
BUILD-MAX-HEAP(A)
1  A·tamanho-do-heap = A·comprimento
2  for i = [comprimento[A]/2] downto 1
3      MAX-HEAPIFY(A, i)
```

A Figura 6.3 mostra um exemplo da ação de BUILD-MAX-HEAP.

Para mostrar por que BUILD-MAX-HEAP funciona corretamente, usamos o seguinte invariante de laço:

No começo de cada iteração do laço **for** das linhas 2-3, cada nó $i + 1$, $i + 2$, …, n é a raiz de um heap de máximo.

Precisamos mostrar que esse invariante é verdadeiro antes da primeira iteração do laço, que cada iteração do laço mantém o invariante e que o invariante dá uma propriedade útil para mostrar a correção quando o laço termina.

Inicialização: Antes da primeira iteração do laço, $i = n/2$. Cada nó $n/2 + 1$, $n/2 + 2$, ..., n é uma folha, e é portanto a raiz de um heap de máximo trivial.

Manutenção: Para ver que cada iteração mantém o invariante de laço, observe que os filhos do nó i são numerados com valores mais altos que i. Assim, pelo invariante de laço, ambos são raízes de heaps de máximo. Essa é precisamente a condição exigida para a chamada MAX-HEAPIFY(A, i) para fazer do nó i uma raiz de heap de máximo. Além disso, a chamada a MAX-HEAPIFY preserva a propriedade de que os nós $i + 1$, $i + 2$, ..., n são raízes de heaps de máximo. Decrementar i na atualização do laço **for** restabelece o invariante de laço para a próxima iteração.

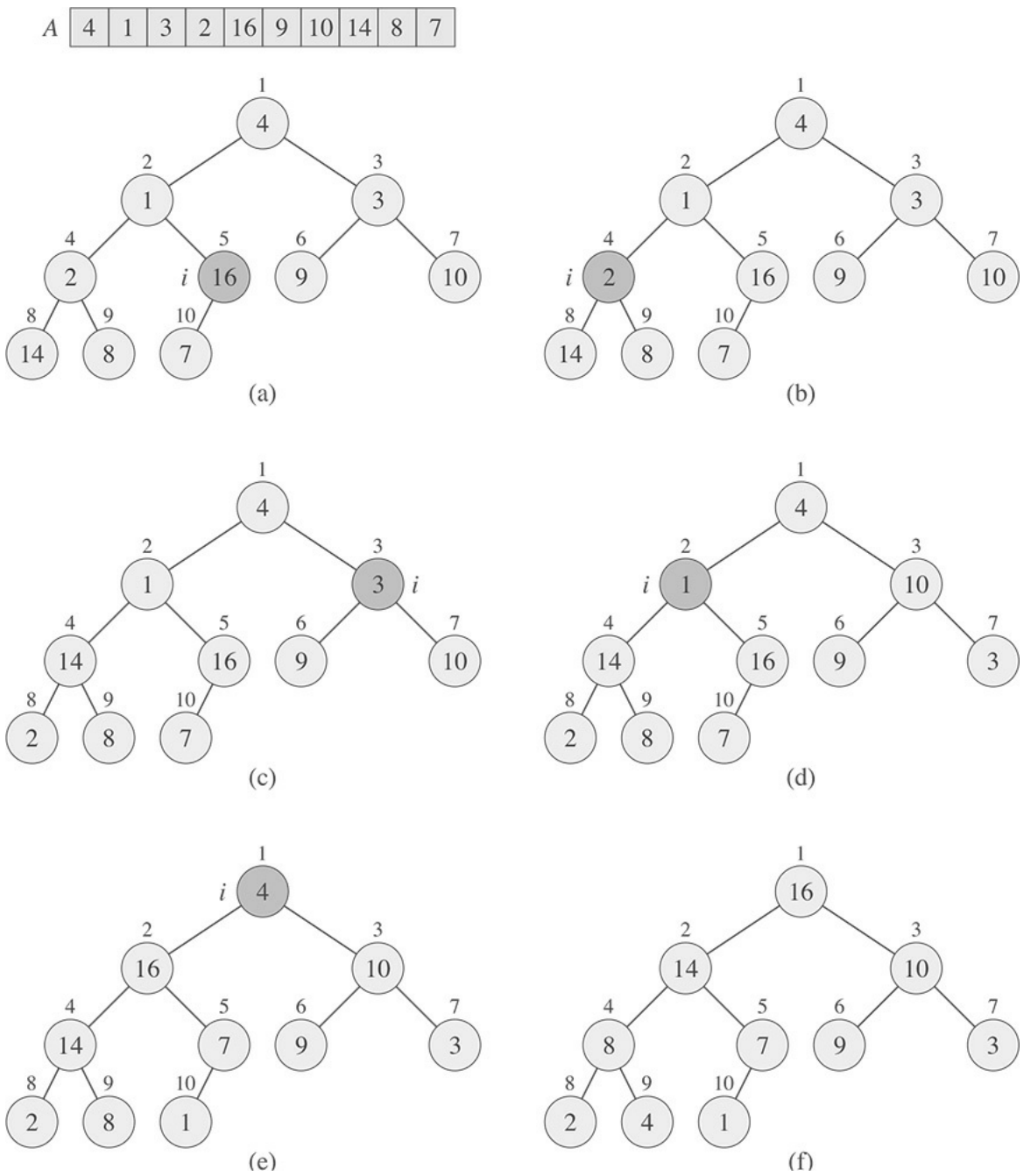

Figura 6.3 Operação de BUILD-MAX-HEAP, mostrando a estrutura de dados antes da chamada a MAX-HEAPIFY na linha 3 de BUILD-MAX-HEAP. (**a**) Um arranjo de entrada de 10 elementos A e a árvore binária que ele representa. A figura mostra que o índice de laço i se refere ao nó 5 antes da chamada MAX-HEAPIFY(A, i). (**b**) A estrutura de dados resultante. O índice de laço i para a próxima iteração aponta para o nó 4. (**c**)–(**e**) Iterações subsequentes do laço **for** em BUILD-MAX-HEAP. Observe que, sempre que MAX-HEAPIFY é chamado em um nó, as duas subárvores desse nó são heaps de máximo. (**f**) O heap de máximo após o término de BUILD-MAX-HEAP.

Término: No término, $i = 0$. Pelo invariante de laço, cada nó 1, 2, ..., n é a raiz de um heap de máximo. Em particular, o nó 1 é uma raiz.

Podemos calcular um limite superior simples para o tempo de execução de BUILD-MAX-HEAP da seguinte maneira: cada chamada a MAX-HEAPIFY custa o tempo $O(\lg n)$, e BUILD-MAX-HEAP faz $O(n)$ dessas chamadas. Assim, o tempo de execução é $O(n \lg n)$. Esse limite superior, embora correto, não é assintoticamente restrito.

Podemos derivar um limite mais restrito observando que o tempo de execução de MAX-HEAPIFY em um nó varia com a altura do nó na árvore, e as alturas na maioria dos nós são pequenas. Nossa análise mais restrita se baseia nas

propriedades de que um heap de n elementos tem altura $\lg n$ (veja Exercício 6.1-2) e, no máximo $n/2^{h+1}$, nós de qualquer altura h (veja Exercício 6.3-3).

O tempo exigido por MAX-HEAPIFY quando chamado em um nó de altura h é $O(h)$; assim, podemos expressar o custo total de BUILD-MAX-HEAP limitado por cima por

$$\sum_{h=0}^{\lfloor \lg n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil O(h) = O\left(n \sum_{h=0}^{\lfloor \lg n \rfloor} \frac{h}{2^h} \right).$$

Avaliamos o último somatório substituindo $x = 1/2$ na fórmula (A.8), o que produz

$$\begin{aligned} \sum_{h=0}^{\infty} \frac{h}{2^h} &= \frac{1/2}{(1-1/2)^2} \\ &= 2. \end{aligned}$$

Assim, podemos limitar o tempo de execução de BUILD-MAX-HEAP como

$$\begin{aligned} O\left(n \sum_{h=0}^{\lfloor \lg n \rfloor} \frac{h}{2^h} \right) &= O\left(n \sum_{h=0}^{\infty} \frac{h}{2^h} \right) \\ &= O(n). \end{aligned}$$

Consequentemente, podemos construir um heap de máximo a partir de um arranjo não ordenado em tempo linear.

Podemos construir um heap de mínimo pelo procedimento BUILD-MIN-HEAP, que é igual a BUILD-MAX-HEAP, a não ser pela chamada a MAX-HEAPIFY na linha 3, que é substituída por uma chamada a MIN-HEAPIFY (veja Exercício 6.2-2). BUILD-MIN-HEAP produz um heap de mínimo a partir de um arranjo linear não ordenado em tempo linear.

Exercícios

6.3-1 Usando a Figura 6.3 como modelo, ilustre a operação de BUILD-MAX-HEAP no arranjo $A = \langle 5, 3, 17, 10, 84, 19, 6, 22, 9 \rangle$.

6.3-2 Por que queremos que o índice de laço i na linha 2 de BUILD-MAX-HEAP diminua de $A \cdot comprimento/2$ até 1, em vez de aumentar de 1 até $A \cdot comprimento/2$?

6.3-3 Mostre que existem, no máximo, $n/2^{h+1}$ nós de altura h em qualquer heap de n elementos.

6.4 O ALGORITMO DE ORDENAÇÃO POR HEAP

O algoritmo de ordenação por heap começa usando BUILD-MAX-HEAP para construir um heap de máximo no arranjo de entrada $A[1..n]$, onde $n = A \cdot comprimento$. Visto que o elemento >máximo do arranjo está armazenado na raiz $A[1]$, podemos colocá-lo em sua posição final correta trocando-o por $A[n]$. Se agora descartarmos o nó n do heap —

e para isso basta simplesmente decrementar $A \cdot comprimento$) —, observaremos que $A[1 .. (n - 1)]$ pode ser facilmente transformado em um heap de máximo. Os filhos da raiz continuam sendo heaps de máximo, mas o novo elemento raiz pode violar a propriedade de heap de máximo. Porém, para restabelecer a propriedade de heap de máximo, basta chamar MAX-HEAPIFY(A, 1), que deixa um heap de máximo em $A[1 .. (n - 1)]$. Então, o algoritmo de ordenação por heap repete esse processo para o heap de máximo de tamanho $n - 1$ até um heap de tamanho 2. (Veja no Exercício 6.4-2 um invariante de laço preciso.)

```
HEAPSORT(A)
1 BUILD-MAX-HEAP(A)
2 for i = comprimento[A] downto 2
3    trocar A[1] com A[i]
4       A·tamanho-do-heap = A·tamanho-do-heap – 1
5       MAX-HEAPIFY(A, 1)
```

A Figura 6.4 mostra um exemplo da operação de HEAPSORT após a linha 1 ter construído o heap de máximo inicial. A figura mostra o heap de máximo antes da primeira iteração do laço **for** das linhas 2–5 e após cada iteração.

O procedimento HEAPSORT demora o tempo $O(n \lg n)$, já que a chamada a BUILD-MAX-HEAP demora o tempo $O(n)$, e cada uma das $n - 1$ chamadas a MAX-HEAPIFY demora o tempo $O(\lg n)$.

Exercícios

6.4-1 Usando a Figura 6.4 como modelo, ilustre a operação de HEAPSORT sobre o arranjo $A = \langle 5, 13, 2, 25, 7, 17, 20, 8, 4 \rangle$.

6.4-2 Discuta a correção de HEAPSORT usando o seguinte invariante de laço:

No início de cada iteração do laço **for** das linhas 2- 5, o subarranjo $A[1 .. i]$ é um heap de máximo que contém os i menores elementos de $A[1 .. n]$, e o subarranjo $A[i + 1 .. n]$ contém os $n - i$ maiores elementos de $A[1 .. n]$, ordenados.

6.4-3 Qual é o tempo de execução de HEAPSORT para um arranjo A de comprimento n que já está ordenado em ordem crescente? E em ordem decrescente?

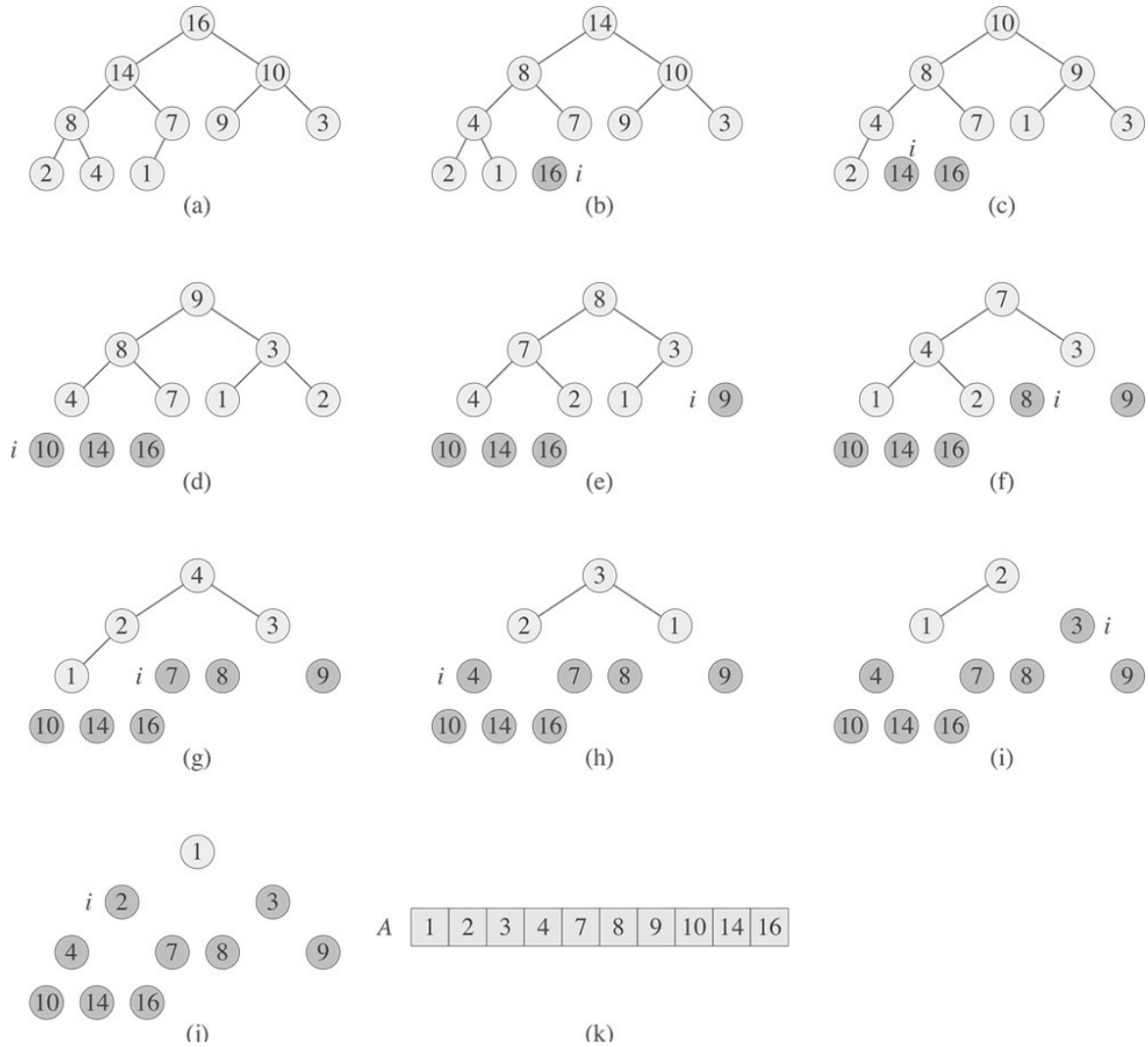

Figura 6.4 Operação de HEAPSORT. (**a**) A estrutura de dados de heap de máximo logo após ter sido construída por BUILD-MAX-HEAP na linha 1. (**b**)–(**j**) O heap de máximo logo após cada chamada de MAX-HEAPIFY na linha 5, mostrando o valor de i nesse instante. Apenas os nós sombreados em tom mais claro permanecem no heap. (**k**) Arranjo ordenado resultante A.

6.4-4 Mostre que o tempo de execução do pior caso de HEAPSORT é $\Omega(n \lg n)$.

6.4-5 ★

Mostre que, quando todos os elementos são distintos, o tempo de execução do melhor caso de HEAPSORT é $\Omega(n \lg n)$.

6.5 FILAS DE PRIORIDADES

A ordenação por heap é um algoritmo excelente, mas uma boa implementação de quicksort, apresentada no Capítulo 7, normalmente o supera na prática. Não obstante, a estrutura de dados de heap propriamente dita tem muitas utilidades. Nesta seção, apresentaremos uma das aplicações mais populares de um heap: uma fila de prioridades eficiente. Como ocorre com os heaps, existem dois tipos de filas de prioridades: filas de prioridade máxima e filas de prioridade mínima. Focalizaremos aqui a implementação de filas de prioridade máxima, que por sua vez se baseiam em heaps de máximo; o Exercício 6.5-3 pede que você escreva os procedimentos correspondentes para filas de prioridade mínima.

Uma ***fila de prioridade*** é uma estrutura de dados para manter um conjunto S de elementos, cada qual com um valor associado denominado ***chave***. Uma ***fila de prioridade máxima*** suporta as seguintes operações:

INSERT(S, x) insere o elemento x no conjunto S. Essa operação é equivalente à operação $S = S \cup \{x\}$.

MAXIMUM(S) devolve o elemento de S que tenha a maior chave.

EXTRACT-MAX(S) remove e devolve o elemento de S que tenha a maior chave.

INCREASE-KEY(S, x, k) aumenta o valor da chave do elemento x até o novo valor k, que admite-se ser, pelo menos, tão grande quanto o valor da chave atual de x.

Entre outras aplicações, podemos usar filas de prioridade máxima para programar trabalhos em um computador compartilhado. A fila de prioridade máxima mantém o controle dos trabalhos a executar e suas prioridades relativas. Quando um trabalho termina ou é interrompido, o escalonador seleciona o trabalho de prioridade mais alta entre os trabalhos pendentes chamando EXTRACT-MAX. O escalonador pode acrescentar um novo trabalho à fila em qualquer instante chamando INSERT.

Alternativamente, uma ***fila de prioridade mínima*** suporta as operações INSERT, MINIMUM, EXTRACT-MIN e DECREASE-KEY. Uma fila de prioridade mínima pode ser usada em um simulador orientado a eventos. Os itens na fila são eventos a simular, cada qual com um instante de ocorrência associado que serve como sua chave. Os eventos devem ser simulados na ordem de seu instante de ocorrência porque a simulação de um evento pode provocar outros eventos a simular no futuro. O programa de simulação chama EXTRACT-MIN em cada etapa para escolher o próximo evento a simular. À medida que novos eventos são produzidos, o simulador os insere na fila de prioridade mínima chamando INSERT. Veremos outros usos de filas de prioridade mínima destacando a operação DECREASE-KEY, nos Capítulos 23 e 24.

Não é nenhuma surpresa que possamos usar um heap para implementar uma fila de prioridade. Em determinada aplicação, como programação de trabalhos ou simulação orientada a eventos, os elementos de uma fila de prioridade correspondem a objetos na aplicação. Muitas vezes, é necessário determinar qual objeto de aplicação corresponde a um dado elemento de fila de prioridade e vice-versa. Portanto, quando usamos um heap para implementar uma fila de prioridade, frequentemente precisamos armazenar um ***descritor*** para o objeto de aplicação correspondente em cada elemento do heap. A constituição exata do descritor (isto é, um ponteiro ou um inteiro) depende da aplicação. De modo semelhante, precisamos armazenar um descritor para o elemento do heap correspondente em cada objeto de aplicação. Nesse caso, normalmente o descritor é um índice de arranjo. Como os elementos do heap mudam de posição dentro do arranjo durante operações de heap, uma implementação real, ao reposicionar um elemento do heap, também teria de atualizar o índice do arranjo no objeto de aplicação correspondente. Visto que os detalhes de acesso a objetos de aplicação dependem muito da aplicação e de sua implementação, não os examinaremos aqui; observaremos apenas que, na prática, esses descritores precisam ser mantidos corretamente.

Agora discutiremos como implementar as operações de uma fila de prioridade máxima. O procedimento HEAP-MAXIMUM implementa a operação Maximum no tempo Q(1).

```
HEAP-MAXIMUM(A)
1 return A[1]
```

O procedimento HEAP-EXTRACT-MAX implementa a operação EXTRACT-MAX. Ele é semelhante ao corpo do laço **for** (linhas 3-5) do procedimento HEAPSORT.

```
HEAP-EXTRACT-MAX(A)
1 if A·tamanho-do-heap < 1
2     error "heap underflow"
3 max = A[1]
4 A[1] = A[A·tamanho-do-heap]
5 A·tamanho-do-heap = A·tamanho-do-heap – 1
6 MAX-HEAPIFY(A, 1)
7 return max
```

O tempo de execução de HEAP-EXTRACT-MAX é $O(\lg n)$, já que ele executa apenas uma quantidade constante de trabalho além do tempo $O(\lg n)$ para MAX-HEAPIFY.

O procedimento HEAP-INCREASE-KEY implementa a operação INCREASE-KEY. Um índice i para o arranjo identifica o elemento da fila de prioridade cuja chave queremos aumentar. Primeiro, o procedimento atualiza a chave do elemento $A[i]$ para seu novo valor. Visto que aumentar a chave de $A[i]$ pode violar a propriedade de heap de máximo, o procedimento, de um modo que lembra o laço de inserção (linhas 5-7) de INSERTION-SORT da Seção 2.1, percorre um caminho simples desde esse nó até a raiz para encontrar um lugar adequado para a chave recém-aumentada. Enquanto HEAP-INCREASE-KEY percorre esse caminho, compara repetidamente um elemento a seu pai, permutando suas chaves, prossegue se a chave do elemento for maior e termina se a chave do elemento for menor, visto que a propriedade de heap de máximo agora é válida. (Veja no Exercício 6.5-5 um invariante de laço preciso.)

```
HEAP-INCREASE-KEY(A, i, chave)
1 if chave < A[i]
2     error "nova chave é menor que chave atual"
3 A[i] = chave
4 while i > 1 e A[PARENT(i)] < A[i]
5       troca A[i] com A[PARENT(i)]
6           i = PARENT(i)
```

A Figura 6.5 mostra um exemplo de operação HEAP-INCREASE-KEY. O tempo de execução de HEAP-INCREASE-KEY para um heap de n elementos é $O(\lg n)$, visto que o caminho traçado desde o nó atualizado na linha 3 até a raiz tem comprimento $O(\lg n)$.

O procedimento MAX-HEAP-INSERT implementa a operação INSERT. Toma como entrada a chave do novo elemento a ser inserido no heap de máximo A. Primeiro, o procedimento expande o heap de máximo, acrescentando à árvore uma nova folha cuja chave é $-\infty$. Em seguida, chama HEAP-INCREASE-KEY para ajustar a chave desse novo nó em seu valor correto e manter a propriedade de heap de máximo.

```
MAX-HEAP-INSERT(A, chave)
1 A·tamanho-do-heap = A·tamanho-do-heap + 1
2 A[A·tamanho-do-heap] = −∞
3 HEAP-INCREASE-KEY(A, A·tamanho-do-heap, chave)
```

O tempo de execução de MAX-HEAP-INSERT para um heap de n elementos é $O(\lg n)$.

Resumindo, um heap pode suportar qualquer operação de fila de prioridade em um conjunto de tamanho n no tempo $O(\lg n)$.

Exercícios

6.5-1 Ilustre a operação de HEAP-EXTRACT-MAX sobre o heap $A = \langle 15, 13, 9, 5, 12, 8, 7, 4, 0, 6, 2, 1 \rangle$.

6.5-2 Ilustre a operação de MAX-HEAP-INSERT(A, 10) sobre o heap $A = \langle 15, 13, 9, 5, 12, 8, 7, 4, 0, 6, 2, 1 \rangle$.

6.5-3 Escreva pseudocódigos para os procedimentos HEAP-MINIMUM, HEAP-EXTRACT-MIN, HEAP-DECREASE-KEY e MIN-HEAP-INSERT que implementem uma fila de prioridade mínima com um heap de mínimo.

6.5-4 Por que nos preocupamos em definir a chave do nó inserido como $-\infty$ na linha 2 de MAX-HEAP-INSERT quando a nossa próxima ação é aumentar sua chave para o valor desejado?

6.5-5 Demonstre a correção de HEAP-INCREASE-KEY usando o seguinte invariante de laço:

No início de cada iteração do laço **while** das linhas 4–6, o subarranjo $A[1 .. A \cdot tamanho\text{-}do\text{-}heap]$ satisfaz a propriedade de heap de máximo, exceto que pode haver uma violação: $A[i]$ pode ser maior que $A[\text{PARENT}(i)]$.

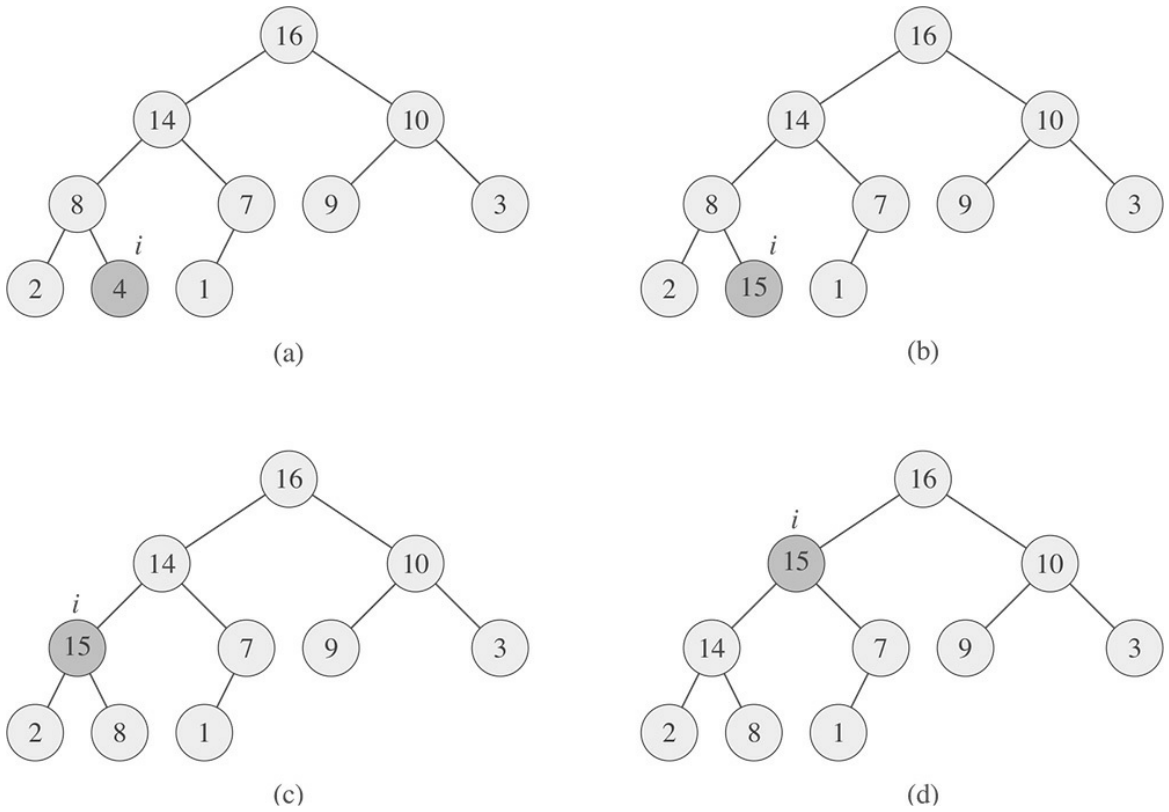

Figura 6.5 Operação de HEAP-INCREASE-KEY. (**a**) O heap de máximo da Figura 6.4(a) com um nó cujo índice é i em sombreado de tom mais escuro. (**b**) A chave desse nó é aumentada para 15. (**c**) Depois de uma iteração do laço **while** das linhas 4–6, o nó e seu pai trocaram chaves, e o índice i sobe para o pai. **(d)** Heap de máximo após mais uma iteração do laço **while**. Nesse ponto, $A[\text{PARENT}(i)] \geq A[i]$. Agora, a propriedade de heap de máximo é válida e o procedimento termina.

Você pode supor que o subarranjo A[1 .. $A \cdot tamanho\text{-}do\text{-}heap$] satisfaz a propriedade de heap de máximo no instante em que HEAP-INCREASE-KEY é chamado.

6.5-6 Cada operação de troca na linha 5 de HEAP-INCREASE-KEY normalmente, requer três atribuições. Mostre como usar a ideia do laço interno de INSERTION-SORT para reduzir as três atribuições a apenas uma atribuição.

6.5-7 Mostre como implementar uma fila primeiro a entrar, primeiro a sair com uma fila de prioridade. Mostre como implementar uma pilha com uma fila de prioridade. (Filas e pilhas são definidas na Seção 10.1.)

6.5-8 A operação HEAP-DELETE(A, i) elimina o item no nó i do heap A. Dê uma implementação de HEAP-DELETE que seja executada no tempo $O(\lg n)$ para um heap de máximo de n elementos.

6.5-9 Dê um algoritmo de tempo $O(n \lg k)$ para intercalar k listas ordenadas em uma única lista ordenada, onde n é o número total de elementos em todas as listas de entrada. (*Sugestão*: Use um heap de mínimo para fazer a intercalação de k entradas.)

Problemas

6-1 Construir um heap com a utilização de inserção

Podemos construir um heap chamando repetidamente MAX-HEAP-INSERT para inserir os elementos no heap. Considere a seguinte variação do procedimento BUILD-MAX-HEAP:

```
BUILD-MAX-HEAP′(A)
1 A·tamanho-do-heap = 1
2 for i = 2 to A·comprimento
3     MAX-HEAP-INSERT(A, A[i])
```

a. Os procedimentos BUILD-MAX-HEAP e BUILD-MAX-HEAP′ sempre criam o mesmo heap quando são executados sobre o mesmo arranjo de entrada? Prove que isso ocorre ou, então, dê um contraexemplo.

b. Mostre que, no pior caso, BUILD-MAX-HEAP′ requer o tempo $\Omega(n \lg n)$ para construir um heap de n elementos.

6-2 Análise de heaps d-ários

Um ***heap d-ário*** é semelhante a um heap binário, mas (com uma única exceção possível) nós que não são folhas têm d filhos em vez de dois filhos.

a. Como você representaria um heap d-ário em um arranjo?

b. Qual é a altura de um heap d-ário de n elementos em termos de n e d?

c. Dê uma implementação eficiente de EXTRACT-MAX em um heap de máximo d-ário. Analise seu tempo de execução em termos de d e n.

d. Dê uma implementação eficiente de INSERT em um heap de máximo d-ário. Analise seu tempo de execução em termos de d e n.

e. Dê uma implementação eficiente de INCREASE-KEY(A, i, k), que sinaliza um erro se $k < A[i]$ mas, caso contrário, ajusta $A[i] = k$ e então atualiza adequadamente a estrutura do heap de máximo d-ário. Analise seu tempo de execução em termos de d e n.

6-3 Quadros de Young

Um ***quadro de Young*** $m \times n$ é uma matriz $m \times n$, tal que as entradas de cada linha estão em sequência ordenada da esquerda para a direita, e as entradas de cada coluna estão em sequência ordenada de cima para baixo. Algumas das entradas de um quadro de Young podem ser ∞, que tratamos como elementos inexistentes. Assim, um quadro de Young pode ser usado para conter $r \leq mn$ números finitos.

a. Trace um quadro de Young 4×4 contendo os elementos $\{9, 16, 3, 2, 4, 8, 5, 14, 12\}$.

b. Demonstre que um quadro de Young $m \times n$ Y é vazio se $Y[1, 1] = \infty$. Demonstre que Y é cheio (contém mn elementos) se $Y[m, n] < \infty$.

c. Dê um algoritmo para implementar EXTRACT-MIN em um quadro de Young $m \times n$ não vazio que é executado no tempo $O(m + n)$. Seu algoritmo deve usar uma subrotina recursiva que resolve um problema $m \times n$ resolvendo recursivamente um subproblema $(m - 1) \times n$ ou um subproblema $m \times (n - 1)$. (*Sugestão*: Pense em MAX-HEAPIFY.) Defina $T(p)$, onde $p = m + n$, como o tempo de execução máximo de EXTRACT-MIN em qualquer quadro de Young $m \times n$. Dê e resolva uma recorrência para $T(p)$ que produza o limite de tempo $O(m + n)$.

d. Mostre como inserir um novo elemento em um quadro de Young $m \times n$ não cheio no tempo $O(m + n)$.

e. Sem usar nenhum outro método de ordenação como subrotina, mostre como utilizar um quadro de Young $n \times n$ para ordenar n^2 números no tempo $O(n^3)$.

f. Dê um algoritmo de tempo $O(m + n)$ para determinar se um dado número está armazenado em determinado quadro de Young $m \times n$.

NOTAS DO CAPÍTULO

O algoritmo de ordenação por heap foi criado por Williams [357], que também descreveu como implementar uma fila de prioridades com um heap. O procedimento BUILD-MAX-HEAP foi sugerido por Floyd [106].

Usamos heaps de mínimo para implementar filas de prioridade mínima nos Capítulos 16, 23 e 24. Também damos uma implementação com limites de tempo melhorados para certas operações no Capítulo 19 e, considerando que as chaves são escolhidas de um conjunto limitado de inteiros não negativos, no Capítulo 20.

Quando os dados são inteiros de b bits e a memória do computador consiste em palavras endereçáveis de b bits, Fredman e Willard [115] mostraram como implementar MINIMUM no tempo $O(1)$ e INSERT e EXTRACT-MIN no tempo $O(\sqrt{\lg n})$. Thorup [337] melhorou o limite $O(\sqrt{\lg n})$ para o tempo $O(\sqrt{\lg n})$. Esse limite usa quantidade de espaço ilimitada em n, mas pode ser implementado em espaço linear com a utilização de hashing aleatorizado.

Um caso especial importante de filas de prioridades ocorre quando a sequência de operações de EXTRACT-MIN é ***monotônica,*** isto é, os valores retornados por operações sucessivas de EXTRACT-MIN são monotonicamente crescentes com o tempo. Esse caso surge em várias aplicações importantes, como o algoritmo de caminhos mais curtos de fonte única de Dijkstra, que discutiremos no Capítulo 24, e na simulação de eventos discretos. Para o algoritmo de Dijkstra, é particularmente importante que a operação DECREASE-KEY seja implementada eficientemente.

No caso monotônico, se os dados são inteiros na faixa 1, 2, ..., C, Ahuja, Melhorn, Orlin e Tarjan [8] descrevem como implementar EXTRACT-MIN e INSERT no tempo amortizado $O(\lg C)$ (consulte o Capítulo 17 para obter mais informações sobre análise amortizada) e DECREASE-KEY no tempo $O(1)$, usando uma estrutura de dados denominada heap digital. O limite $O(\lg C)$ pode ser melhorado para $O(\sqrt{\lg C})$ com a utilização de heaps de Fibonacci (consulte o Capítulo 19) em conjunto com heaps digitais. Cherkassky, Goldberg e Silverstein [65] melhoraram ainda mais o limite até o tempo esperado $O(\lg^{1/3 + \varepsilon} C)$ combinando a estrutura de baldes em vários níveis de Denardo e Fox [85] com o heap de Thorup já mencionado. Raman [291] aprimorou mais ainda esses resultados para obter um limite de $O(\min(\lg^{1/4 + \varepsilon} C, \lg^{1/3 + \varepsilon} n))$, para qualquer $\varepsilon > 0$ fixo.